**Freescale Semiconductor**
Application Note

Document Number: AN3896
Rev. 0, 10/2009

# MCF51EM256 Performance Assessment with Algorithms Used in Metering Applications

by: Paulo Knirsch
MSG IMM System and Applications

## 1 Introduction

This application note's objective is to demonstrate the implementation of the following algorithms used in metering applications using Freescale MCF51EM256.

- Square root
- Voltage and current RMS values
- Active energy, active power, apparent power, reactive power, and power factor
- Discrete fourier transform (DFT)
- Total harmonic distortion (THD)

The methodology used is to present the formula for the calculation, discuss its implementation, and present execution performance analysis.

The calculation of the performance required to process the algorithms is estimated. The following assumptions are made:

- The power line frequency is 60 Hz.

**Contents**





freescale semiconductor

- The sampling rate is 15.360 kHz, therefore there are 256 samples per power line cycle.
- 256 samples are stored in a double buffer, while one buffer is being updated the other is used for the calculations.

The MCF51EM256 was configured to operate with 25 MHz of bus frequency.

# 2 Test Setup

The DEMOEM demo board was used. The test software project, EM256_Performance.mcp was developed with CodeWarrior V6.2. The compiler optimization was set to Level 4, faster execution speed. The tests were executed with the CodeWarrior debugger using the Terminal Window application from P&E Micro to visualize the results (configure to 19200 bps, parity none, and 8-bits).

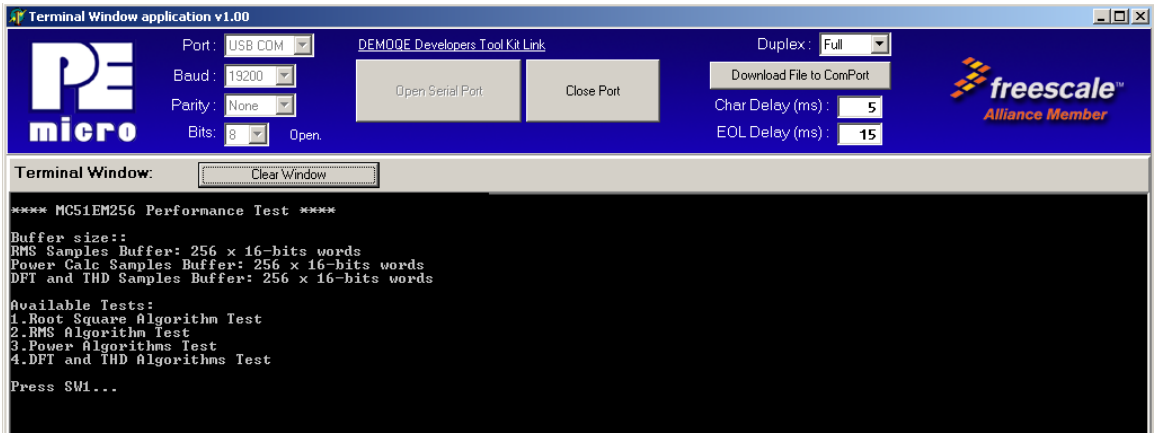


**Figure 1. Terminal window application**

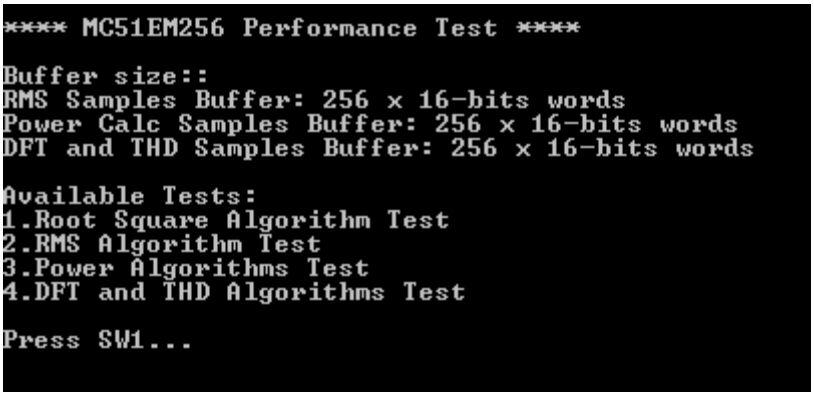


**Figure 2. MCF51EM256 performance test initial window**

The test project main files are illustrated in Figure 3.

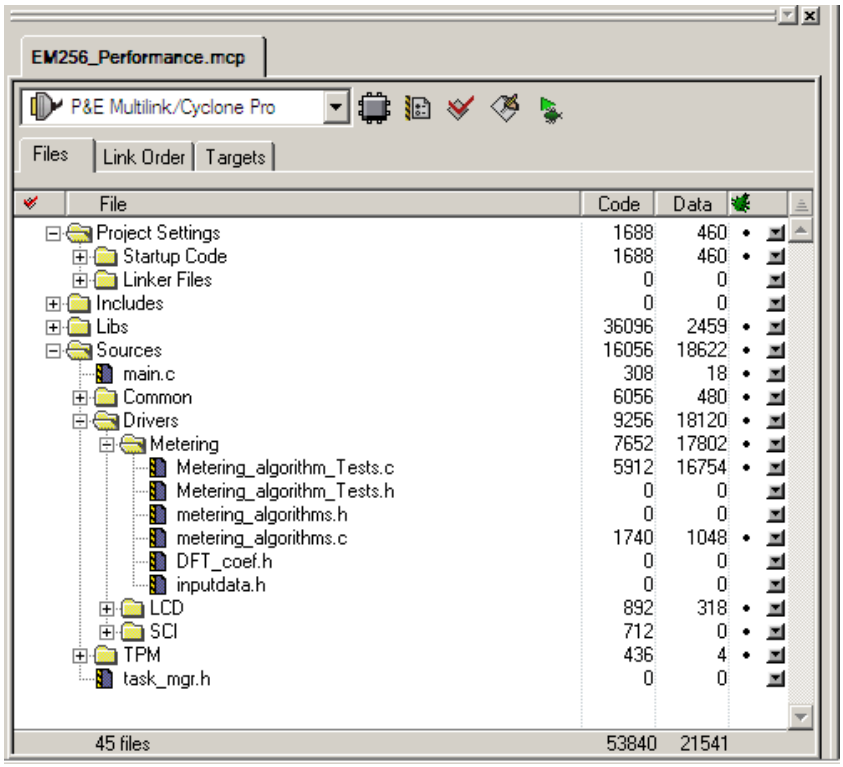


**Figure 3. Project structure**

The metering tests are implemented in the files:

- Metering_algorithm_Tests.c
- Metering_algorithm_Tests.h

The algorithms tests are implemented in the files:

- Metering_algorithms.c
- Metering_algorithms.h

The DFT coefficients are in the file:

- DFT_coef.h

The input data sets used to perform the test are in the file:

- inputdata.h

Table 1 describes the datasets used for the tests.

**Table 1. Input datasets**

| Data set name | Vector size | Variable type | Data description |
|---|---|---|---|
| InputVec | 256 | unsigned 32-bit | 0, 1, 2, 3, ... ,254, 255 |
| InputVec1 | 256 | unsigned 32-bit | Random numbers with uniform distribution in the range of 0 to 65535 (16-bits) |
| InputVec2 | 256 | unsigned 32-bit | Random numbers with uniform distribution in the range of 0 to $2^{32}$ –1 |
| InputVec3 | 256 | unsigned 32-bit | Sequential incremental numbers from ($2^{32}$ –256) to ($2^{32}$ –1) |
| InputVecRMS | 256 | signed 16-bits | 60 Hz, 32767 amplitude, 0 phase sine wave |
| InputVecRMS1 | 256 | signed 16-bits | 60 Hz, 1000 amplitude, 0 phase sine wave |
| InputVecRMS2 | 256 | signed 16-bits | 60 Hz, 100 amplitude, 0 phase sine wave |
| InputVecRMS3 | 256 | signed 16-bits | 60 Hz, 10 amplitude, 0 phase sine wave |
| inputsignal1 | 256 | signed 16-bits | 60 Hz, 32767 amplitude, 0 phase sine wave |
| inputsignal2 | 256 | signed 16-bits | 60 Hz, 32767 amplitude, 45° phase sine wave |
| inputsignal3 | 256 | signed 16-bits | 45 Hz, 16000 amplitude, 0 phase sine wave |
| inputsignal4 | 256 | signed 16-bits | 75 Hz, 16000 amplitude, 0 phase sine wave |
| inputsignal5 | 256 | signed 16-bits | 60 Hz, 2000 amplitude, 0 phase sine wave + 180 Hz, 16000 amplitude, 0 phase sine wave |

The expected outputs of the algorithms were determined using the excel spreadsheet Performance Analysis.xls. They were obtained doing exactly the same algorithm as implemented in the C code for the MCU.

# 3 Square Root

Formula

The square root is calculated using the Babylonian method, see Equation 1.

$$x_0 \approx \sqrt{S}$$
$$x_0 + 1 = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right),$$
$$\sqrt{S} = \lim_{n \to \infty} x_n$$

*Eqn. 1*

This method calculates the square root by an interaction. The seed value X0 needs to be near the desired square root value to reduce the number of interactions necessary for a good precision result.

The example implementation here uses the following seed value for the square root calculations. Equation 2 is a square root initial guess.

$$X0 = 2^{[D/2]} \text{ (here D is the number of binary digits)}$$

*Eqn. 2*

## 3.1 Implementation

```
//---------------------------------------------
//   Square root
//
//   Input parameter: unsigned 32-bits
//   Output parameter: unsigned 16-bits
//---------------------------------------------
/*Square root*/
word  SquareRoot(dword A){

  word j;                           // local variable
  Acc = ASM_FF1(A);                 // initial seed
  for(j=0;j<MAX_INT;j++){           // execute maximum of MAX_INT interactions
    Acc_temp = (A/Acc + Acc)/2;     // Calculate interaction
    if((abs(Acc - Acc_temp) < 0x0001)||(Acc_temp == 0)) // Test is precision is good enough
      break;                        // break if a good precision was reached
    Acc = Acc_temp;                 // Save previous interaction
  }
  //Number_of_interactions[i] = (unsigned short) (j+1); // For debugging purposes
  return((word)Acc_temp);           // Return parameter
}
```

**Figure 4. Square root code implementation**

```
//---------------------------------------------
//   Guess the initial root square seed
//
//   Input parameter: unsigned 32-bits
//   Output parameter: unsigned 16-bits
//---------------------------------------------
/*Root Square Initial Seed*/
dword ASM_FF1(dword A){
  asm{                          //
  ff1.l d0                      // find the fist one
  }                             //
  A= (32-A)>>1;                 // first one index diveded by 2
  A = (unsigned long)1<<A;      // multiplies by two
  return(A);                    // Return parameter
}
```

**Figure 5. Square root seed generation**

## 3.2 Square Root Test Result

The algorithm performance was tested with four different data sets.

**Table 2. Datasets used for a square root test**

| Dataset | Dataset name (check table 1) |
|---|---|
| 1st | InputVec |
| 2nd | InputVec1 |
| 3rd | InputVec2 |
| 4th | InputVec3 |

The output of the test obtained with the terminal window can be seen in Figure 6

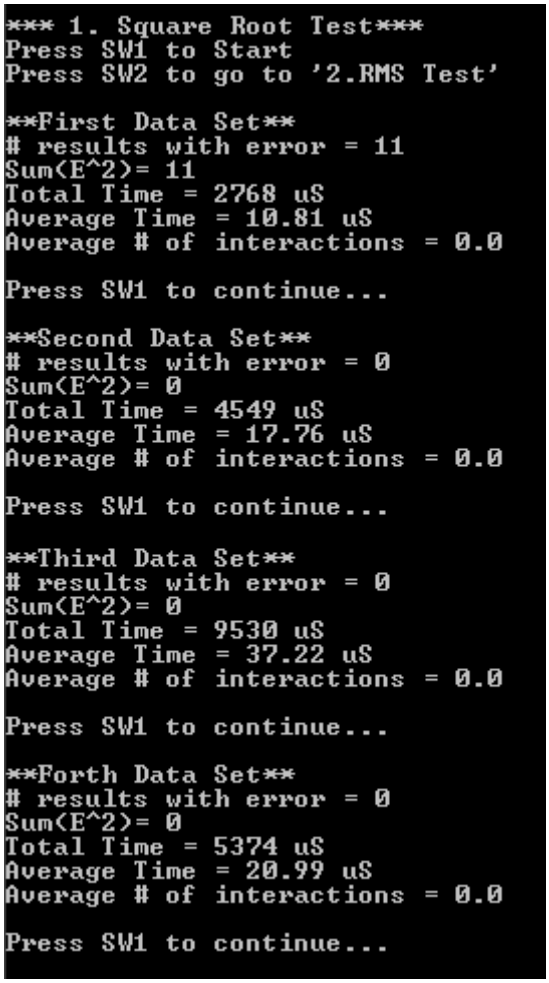


**Figure 6. Square root test terminal window output**

Table 3 summarizes the square root test result.

**Table 3. Square root test result (for QE128)**

| Square Root Test Result Summary (QE128) | | | | | |
|---|---|---|---|---|---|
| | **Description** | **Number of results with error** | **Error standard deviation** | **Average execution time per square root** | **Average number of interactions** |
| **1st Dataset** | values from 0 to 255 | 4 | 0.125 | 11.8 µs | 3.152 |
| **2nd Dataset** | Random number in the range of 0 to 2^16–1 | 0 | 0 | 20.1 µs | 2.738 |
| **3rd Dataset** | Random number in the range of 0 to 2^32 –1 | 0 | 0 | 44.5 µs | 3.723 |
| **4th Dataset** | Numbers from (2^32 – 256) to (2^32 –1) by 1 increments. | 0 | 0 | 24.3 µs | 2.000 |

**Table 4. Square root test result**

| Square Root Test Result Summary | | | | |
|---|---|---|---|---|
| **Dataset** | **Number Results with error** | **Error standard deviation** | **Average execution time per square root** | **Average number interaction** |
| **1st** | 11 | 0.207 | 10.97 µs | 2.27 |
| **2nd** | 0 | 0 | 18.3 µs | 2.73 |
| **3rd** | 0 | 0 | 37.47 µs | 3.72 |
| **4th** | 0 | 0 | 21.33 µs | 2.0 |

Maximum observed time is 48 µs for one 32-bit square root calculation.

# 4 RMS Calculation

The RMS equation is as follows:

$$VRMS = V_{rms} = \frac{1}{N} \times \sum_{i=1}^{N} V^2(i) \qquad \textit{Eqn. 3}$$

## 4.1 Implementation

```
//-------------------------------------------
//   RMS
//
//   Input parameter: pointer to 16-bit signed.
//   Output parameter: unsigned 16-bits.
//-------------------------------------------
/* RMS calculation*/
word  RMS_calc(short *input){

  Sum = 0;                                  // clear accumulator variable
  for(i=0;i<N_SAMPLES;i+=RMS_DEC){          // execute interaction
#if ASM_SUM == 1                            // Assembly implementation of 64-bit sum
      ADD64bits(&Sum,((*input)*(*input)));  // call Assembly function
      input+=RMS_DEC;                       // update pointer
#else                                       // if not using assembly
    Sum = Sum + (input[i])*(input[i]);      // multiply and accumulate
#endif                                      //
  }                                         //
  Sum = (Sum>>FAC_SAMPLE_FOR_RMS);          // divide final sum by number of samples
  return(SquareRoot((dword)Sum));           // output square root of Sum

}
```

**Figure 7. RMS algorithm code implementation**

The intermediate result was stored with 64-bit precision to avoid intermediate results overflow.

The following implementation optimizations were made to reduce the execution time:

- Implementation of a 32-bit addition with 64-bit accumulation in assembly
- Use of pointers instead of arrays
- Division implemented by shifting

Figure 8 shows the 64-bit addition implemented in assembly.

```
//-------------------------------------------
//   64-bit accumulate
//
//   Input parameter: signed 64-bits pointer
//   Output parameter: signed 32-bits
//-------------------------------------------
/*64-bit accumulate*/
asm void ADD64bits(long long* ACC, long A){

          ADD.L  D0,4(A0)                   // Add 32-bit to 64-bit accumulator
          BCC    OUT                        // Check overflow
          ADDQ.L #01,(A0)                   // if overflow increase MS 32-bits of ACC
    OUT:                                    //
          RTS                               // Return
}
```

**Figure 8. 32-bits addition to 64-bit accumulator**

A “define” controls if the algorithms use the standard C compiler 64-bit addition or the proposed assembly implementation. The “define” is in the metering_algorithms.h and is shown below. If defined as 1 it uses the assembly 64-bit addition. If defined as 0 use the standard C addition.

```
# define ASM_SUM 1
```

The RMS implementation allows the configuration of the number of samples to be used for the calculation. The SAMPLE_FOR_RMS define shown below controls and can be found at the top of the metering_algorithms.h file.

```
# define Sample_FOR_RMS 32
```

## 4.2 RMS Test Results

The algorithm performance was tested with four different data sets.

**Table 5. Datasets used for the RMS test**

| Dataset | Data set name (check table 1) |
|---|---|
| 1st | InputVecRMS |
| 2nd | InputVecRMS1 |
| 3rd | InputVecRMS2 |
| 4th | InputVecRMS3 |

Figure 9 illustrates the RMS test terminal window output.

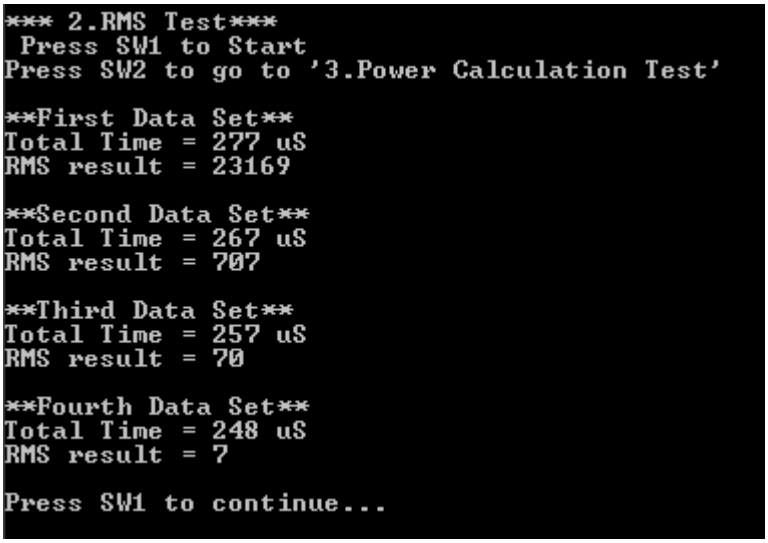


**Figure 9. RMS Test terminal output (# samples = 256)**

The RMS tests were repeated using 256, 128, 64, and 32 samples to evaluate the execution time versus precision trade-off. The results are shown in the Table 5 and Table 6.

**Table 6. RMS execution time**

| RMS Execution Time | | | | |
|---|---|---|---|---|
| **Dataset** | **256 Samples** | **128 Samples** | **64 Samples** | **32 Samples** |
| • 1st | 277 µs | 166 µs | 103 µs | 72 µs |
| • 2nd | 267 µs | 156 µs | 94 µs | 57 µs |
| • 3rd | 257 µs | 145 µs | 83 µs | 52 µs |
| • 4th | 248 µs | 136 µs | 74 µs | 43 µs |

**Table 7. RMS test results precision**

| RMS Test Precision | | | | | |
|---|---|---|---|---|---|
| **Dataset** | **Expected value** | **256 Samples result** | **128 Samples result** | **64 Sample results** | **32 Sample results** |
| **1st** | 23169.77 | 23169 | 23169 | 23169 | 23169 |
| **2nd** | 707.11 | 707 | 707 | 707 | 707 |
| **3rd** | 70.71 | 70 | 70 | 70 | 70 |
| **4th** | 7.07 | 7 | 7 | 7 | 7 |

Observed results are as expected considering the truncation due to quantization.

One additional optimization that can be done is to store the intermediate results in 32-bits precision. This requires normalization of the intermediate values resulting in less precision of the RMS value.

# 5 Power Measurements

Power measurements is a routine that receives as input the voltage buffer, current buffer, and calculates the following outputs:

1. Total energy
2. Active power
3. Reactive power
4. Apparent power
5. Vrms and Irms
6. Power factor

The buffers must contain 256 samples (N) that must correspond to a complete period of the power main.

The outputs are calculated with the following equations:

$$Energy = \sum_{i=0}^{N} V([i] \times I[i]) \quad \textbf{Eqn. 4}$$

$$ActivePower = Energy/N \quad \textbf{Eqn. 5}$$

$$ApparentPower = Vrms \times Irms \quad \textbf{Eqn. 6}$$

$$ReactivePower = \sqrt{ApparentPower^2 - ActivePower^2} \quad \textbf{Eqn. 7}$$

$$Power(Factor) = ActivePower - ApparentPower \quad \textbf{Eqn. 8}$$

Vrms and Irms are calculated as described in the Section 4, “RMS Calculation .”

## 5.1 Implementation

```
 Input parameter: -Voltage pointer to 16-bit signed.
                  -Current pointer to 16-bit signed.
                  -Output Vector pointer.
                  -Output Time vector pointer.
 Output parameter: unsigned 16-bits.
 ----------------------------------------
 Power_Calc(short *V, short *I, Power_vec *Out, Power_time *tm){

 g A;                                    //local var used as ACC
 :t *tempV = V, *tempI = I;              //local pointers
                                         //
 ıctive Energy Calculation               //
 ırtTPM();                               //Start timer to measure execution time
 ı = 0;                                  //clear accumulator variable
 :(i=0;i<N_SAMPLES;i++){                 //interaction for the 256 samples
 Sum = Sum + (*tempV++)*(*tempI++);      //multiply and accumulate
                                         //
 :->Act_Eng = Sum;                       //output result
 ->Act_Eng = Stop_Read_TPM();            //Stop timer to measure execution time
                                         //
 ıctive Power Calculation                //
 ırtTPM();                               //Start timer to measure execution time
 :->Act_Pwr = (long)(Sum>>FAC_N_SAMPLES);//Active Power Calculation
 ->Act_Pwr = Stop_Read_TPM();            //Stop timer to measure execution time
                                         //
 Voltage RMS Calculation                 //
 ırtTPM();                               //Start timer to measure execution time
 :->Vrms = RMS_calc(V);                  //Voltage RMS calculation
 ->Vrms = Stop_Read_TPM();               //Stop timer to measure execution time
                                         //
 Current RMS Calculation                 //
 ırtTPM();                               //Start timer to measure execution time
 :->Irms = RMS_calc(I);                  //Current RMS calculation
 ->Irms = Stop_Read_TPM();               //Stop timer to measure execution time
                                         //
 ıparent Power Calculation               //
 ırtTPM();;                              //Start timer to measure execution time
 :->Apr_Pwr = (long)((Out->Vrms) * (Out->Irms));//Aparent Power Calculation
 (Out->Apr_Pwr) < (Out->Act_Pwr))        //Check if Aparent power is smaller then active
 )ut->Apr_Pwr = Out->Act_Pwr;            //Correct output
 ->Apr_Pwr = Stop_Read_TPM();            //Stop timer to measure execution time
                                         //
 Power Factor Calculation                //
 ırtTPM();                               //Start timer to measure execution time
 (Out->Apr_Pwr)<=(Out->Act_Pwr))         //If aparent power equals active
 )ut->Pwr_fct = 65535;                   //Power Factor equals one
 se                                      //
 )ut->Pwr_fct = (word) ((((long long)(Out->Act_Pwr))<<16)/(Out->Apr_Pwr));//calc P. Fct
 ->Pwr_fct = Stop_Read_TPM();            //Stop timer to measure execution time
                                         //
 Reactive Energy Calculation             //
 ırtTPM();                               //Start timer to measure execution time
 = (((Out->Apr_Pwr)>>16)*((Out->Apr_Pwr)>>16)) - (((Out->Act_Pwr)>>16)*((Out->Act_Pwr)>>16)
 )ut->React_Pwr = (SquareRoot((dword)A))<<16;
 ->React_Pwr = Stop_Read_TPM();          //Stop timer to measure execution time
```

**Figure 10. Power calculation implementation code**

The energy calculation algorithm used 64-bit intermediate results as well as the Vrms and Irms to ensure maximum precision.

**Table 8. Algorithms output data types**

| Output | Data type |
|---|---|
| Total energy | 64-bit signed |
| Active Power | 32-bit signed |
| RMS | 16-bit unsigned |
| Reactive Power[1] | 32-bit signed |
| Power factor[2] | 16-bit unsigned |

[1] Use 16-bit intermediate result. Higher precision results could be achieved with a 32-bit output square root algorithm.

[2] 65536 correspond to “1” power factors.

The implementation includes the StartTPM() and Stop_Read_TPM() functions. These are used for debug purposes only. These lines must be removed in the final implementation.

The output values are stored in the structure defined as shown in Figure 11.

```
/*Metering Datatypes*/
//Power Measures Structure
typedef struct{
  long long Act_Eng;    // Active Energy
  long Act_Pwr;         // Active Power
  long React_Pwr;       // Reactive Power
  long Apr_Pwr;         // Aparent Power
  word Vrms;            // Vrms
  word Irms;            // Irms
  word Pwr_fct;         // Power_factor
}Power_vec;
```

**Figure 11. Power variable structure**

The Power_time structure was defined to store measured execution times for debugging purposes only. It may be removed from the implementation if it is not required.

## 5.2 Test Results

The algorithm performance was tested with three different data sets each one having two 256 16-bit inputs, one used as the voltage buffer, and the other used as the current buffer.

**Table 9. Power calculation test datasets**

| Dataset | Data Set Name (check table 1) | |
|---|---|---|
| | Voltage input | Current Input |
| 1st | InputVecRMS | InputVecRMS |
| 2nd | inputsignal1 | Inputsignal2 |
| 3rd | Inputsignal3 | Inputsignal4 |

```
*** 3.Power Calculation Test***
 Press SW1 to Start
Press SW2 to go to '4.DFT Test'

First Dataset
Active Energy: 137430225637 @609 uS
Active Power: 536836818 @3 uS
React. Power: 0 @3 uS
Apare. Power: 536836818 @1 uS
Vrms: 23169 @72 uS
Irms: 23169 @72 uS
Power Factor: 65535@ 0 uS

Press SW1 to continue...
```

**Figure 12. Terminal window output for power calculation test (dataset 1)**

**Table 10. 1st dataset results table**

| | 1st Dataset Power Calculation Results | | | |
|---|---|---|---|---|
| | expected | obtained | error | execution time [µs] |
| Energy | 137430225637 | 137430225637 | 0.000000% | 609 |
| Act_Pwr | 536836819 | 536836818 | 0.000000% | 3 |
| Vrms | 23170 | 23169 | 0.003191% | 276 |
| Irms | 23170 | 23169 | 0.003191% | 277 |
| Apr_Pwr | 536836819 | 536836818 | 0.000000% | 1 |
| React_Pwr | 0 | 0 | 0.000000% | 3 |
| Prw_fct | 65536 | 65535 | 0.001526% | 0 |
| | | | Total time | 1,169 |

**Table 11. 2nd Dataset results table**

| | 2nd Dataset Power Calculation Results | | | |
|---|---|---|---|---|
| | expected | obtained | error | execution time [uS] |
| Energy | 97178042060 | 97178042060 | 0.0000000% | 596 |
| Act_Pwr | 379601727 | 379601726 | 0.0000003% | 3 |
| Vrms | 23170 | 23169 | 0.0043159% | 263 |
| Irms | 23170 | 23169 | 0.0043159% | 263 |
| Apr_Pwr | 536837910 | 536802561 | 0.0065847% | 0 |
| React_Pwr | 379601727 | 379453440 | 0.0390638% | 34 |
| Prw_fct | 46341 | 46344 | 0.0064737% | 87 |
| | | | Total time | 1,246 |

**Table 12. 3rd Dataset results table**

| | 3rd Dataset Power Calculation Results | | | |
|---|---|---|---|---|
| | expected | obtained | error | execution time [uS] |
| Energy | 127874757 | 127874757 | 0.0000000% | 591 |
| Act_Pwr | 499511 | 499510 | 0.0002002% | 3 |
| Vrms | 11292 | 11291 | 0.0088558% | 259 |
| Irms | 11292 | 11291 | 0.0088558% | 261 |

**Table 12. 3rd Dataset results table (continued)**

| | | | | |
|---|---|---|---|---|
| **Apr_Pwr** | 127500244 | 127486681 | 0.0106376% | 1 |
| **React_Pwr** | 127499265 | 127401984 | 0.0762993% | 24 |
| **Prw_fct** | 257 | 256 | 0.3891051% | 80 |
| | | | Total time | 1,219 |

# 6 Discrete Fourier Transform (DFT)

A popular method used for energy metering is the discrete fourier transform (DFT) that can estimate the voltage and current phasors. At the same time it eliminates the DC component and harmonics. The phasors are described as follows:

DFT formula:

$$Z_{rk} = \frac{2}{N}\sum_{r-0}^{N-1} Z_{k-r}\cos\frac{2\pi r}{N}$$
$$Z_{ik} = \frac{2}{N}\sum_{r-0}^{N-1} Z_{k-r}\sin\frac{2\pi r}{N}$$

***Eqn. 9***

Where $Z_{rk}$ is the phasor's real part, $Z_{ik}$ is the phasor's imaginary part and both are expressed as functions of the k element. $Z_{k-r}$ is the k sample, N is the number of samples, and r is the angle step defined for the function sine and cosine functions.

Using this approach in Equation 10 voltages are expressed with the DFT formula:

$$V_{rk} = \frac{2}{N}\sum_{r-0}^{N-1} V_{k-r}\cos\frac{2\pi r}{N}$$
$$V_{ik} = \frac{2}{N}\sum_{r-0}^{N-1} V_{k-r}\sin\frac{2\pi r}{N}$$

***Eqn. 10***

The voltage phasor is obtained by:

$$|\bar{V}| = \sqrt{V_r^{\,2} + V_i^{\,2}}$$
$$\theta = \tan^{-1}\frac{V_r}{V_i}$$

***Eqn. 11***

The mean and RMS values are obtained by:

$$V_{mean} = |\bar{V}|$$
$$V_{RMS} = \frac{|\bar{V}|}{\sqrt{2}}$$

***Eqn. 12***

The same mathematical operations are used to obtain the current phasor and are not repeated here.

The complex, active, and reactive power can be expressed in terms of the current and voltage phasors as follows.

$$\begin{aligned} S &= VI^* = P + jQ \\ P &= V_r I_r + V_1 I_1 \\ Q &= V_1 I_r - V_r I_1 \end{aligned}$$

**Eqn. 13**

## 6.1 Implementation

The input parameter is a pointer to a 16-bit signed data buffer.

The output parameter is a structure described in Equation 13. It returns the real and imaginary values of the output complex vector. Both the real and imaginary values are stored as a 32-bit signed value (word).

The intermediate result was stored with 64-bit precision to avoid intermediate results overflow. The implementation uses 32-bit intermediate results, if 64-bit values are not needed to reduce the execution time.

```
typedef struct {
  long Real;
  long Img;
} Complex;
```

**Figure 13. Complex structure**

The sample frequency $f_s$ and the number of points in the data buffer, N, determine the fundamental frequency component of the DFT output. The fundamental frequency of the DFT can be calculated using Equation 14.

$$F_k = f_s / N$$

**Eqn. 14**

In the case considered for this application note, $f_s$ = 15.360 kHz and N = 256. Therefore, the $F_k$ frequency is 60 Hz, exactly the power network frequency being considered. Higher frequency harmonics can be calculated using the formula below, DFT formula for “k” harmonic:

$$Z\text{real}_k = \frac{2}{N} \sum_{n=0}^{N-1} X_n \cos\left(\frac{2\pi nk}{N}\right)$$

$$Zimg_k = \frac{2}{N} \sum_{n=0}^{N-1} x_n \sin\left(\frac{2\pi nk}{N}\right)$$

**Eqn. 15**

Where “k” is the harmonic number. In this case, the harmonic frequency is:

$$F_k = f_s * k / N$$

**Eqn. 16**

Then for the network third harmonic, k = 3 and $F_k$ = 180 Hz.

```
//-----------------------------------------
//   DFT
//
//   Input parameter: signed 16-bits pointer
//   Output parameter: Complex Structure
//-----------------------------------------
Complex DFT(short *input){

  Complex Res;                                  // Complex structure to store result
  short *P1, *P2;                               // pointers
  P1= input;                                    // input pointer initialization (Real)
  P2 = &Cos_coef_k_1[0];                        // coeficients pointer init (Real)
                                                //
  Sum_D = 0;                                    // clear accumulator variable
  for(i=0;i<N_SAMPLES;i+=DEC){                  // calculate real component
    Sum_D = Sum_D + (((*P1)*(*P2))>>DFT_SCALING);// square and accumulate the N_SAMPLES_DFT samples
    P1+=DEC;                                    // increment pointers
    P2+=DEC;                                    // increment pointers
  }                                             //
  Res.Real = (long)((Sum_D)>>(FAC_SAMPLE_FOR_DFT + COEF_MAX - 1 - DFT_SCALING));
                                                //
  P1= input;                                    // input pointer initialization (Imag)
  P2 = &Sin_coef_k_1[0];                        // coeficients pointer init (Imag)
                                                //
  Sum_D = 0;                                    // clear accumulator variable
  for(i=0;i<N_SAMPLES;i+=DEC){                  // calculate Img component
     Sum_D = Sum_D + (((*P1)*(*P2))>>DFT_SCALING);// square and accumulate the N=256 buffer samples
     P1+=DEC;                                   // increment pointers
     P2+=DEC;                                   // increment pointers
  }                                             //
  Res.Img =  (long)((Sum_D)>>(FAC_SAMPLE_FOR_DFT + COEF_MAX - 1 - DFT_SCALING));
                                                //
  return(Res);                                  // output the result

}
```

**Figure 14. DFT Implementation for first harmonic**

To allow faster execution speed, the implementation allows selection of the number of samples used for the DFT calculation.

```
#define SAMPLE_FOR_DFT 256 // number of samples used for the DFT
```

The less number of samples used, the faster the algorithm. Using 128 samples or less, (apart from less interactions in the loop), has the additional advantage of storing the intermediate results in 32-bit values.

The implementation for calculating other harmonics require a different Sin_coef_k vector. This vector has to be calculated with Equation 11.

## 6.2 Test Result

The algorithm performance was tested with five different data sets, each one having 256 16-bit inputs.

**Table 13. DFT test datasets**

| Dataset | Data Set Name (check table 1) |
|---|---|
| 1st | Inputsignal1 |
| 2nd | Inputsignal2 |
| 3rd | Inputsignal3 |
| 4th | Inputsignal4 |
| 5th | inputsignal5 |

```
*** 4.DFT Test***
 Press SW1 to Start
Press SW2 to go back to Main Menu

DFT input signal 1: (0 + 32765j) @1318 uS
DFT input signal 1: Fundamental RMS (23168), THD (608) @65 uS

DFT input signal 2: (23169 + 23169j) @1318 uS
DFT input signal 2: Fundamental RMS (23169), THD (0) @43 uS

DFT input signal 3: (-8669 + 11640j) @1312 uS
DFT input signal 3: Fundamental RMS (10262), THD (30072) @72 uS

DFT input signal 4: (11254 + 9054j) @1318 uS
DFT input signal 4: Fundamental RMS (10213), THD (30891) @80 uS

DFT input signal 5: (2586 + 20694j) @1318 uS
DFT input signal 5: Fundamental RMS (14746), THD (27581) @71 uS

Press SW1 to continue...
```

**Figure 15. Terminal window output for the DFT, THD, and fundamental RMS tests**

**Figure 16. Real component calculation with errors**

| DFT Test Precision (Real Component) | | | | | |
|---|---|---|---|---|---|
| **Dataset** | **Expected Value** | **256 Samples Result** | **128 Samples Result** | **64 Samples Result** | **32 Samples Result** |
| **1st** | 0 | 0 (0.0000%) | –1 (–%) | –1 (–%) | –1 (–%) |
| **2nd** | 23169.8 | 23169 (0.0035%) | 23169 (0.0035%) | 23168 (0.0078%) | 23169 (0.0035%) |
| **3rd** | –8668.5 | –8669 (0.0058%) | –8607 (0.7095%) | –8484 (2.1284%) | –8243 (4.9086%) |
| **4th** | 11254.9 | 11254 (0.0080%) | 11191 (0.5678%) | 11062 (1.7139%) | 10797 (4.0685%) |
| **5th** | 2586.4 | 2586 (0.0155%) | 2584 (0.0928%) | 2578 (0.3248%) | 2554 (1.2527%) |

**Figure 17. Imaginary component calculation with errors**

| DFT Test Precision (Imaginary Component) | | | | | |
|---|---|---|---|---|---|
| **Dataset** | **Expected Value** | **256 Samples Result** | **128 Samples Result** | **64 Samples Result** | **32 Samples Result** |
| **1st** | 32767 | 32765 (0.0061%) | 32766 (0.0031%) | 32765 (0.0061%) | 32765 (0.0061%) |
| **2nd** | 23169.8 | 23169 (0.0035%) | 23169 (0.0035%) | 23168 (0.0078%) | 23169 (0.0035%) |
| **3rd** | 11640.8 | 11640 (0.0069%) | 11639 (0.0155%) | 11636 (0.0412%) | 11624 (0.1443%) |
| **4th** | 9054.4 | 9054 (0.0044%) | 9054 (0.0044%) | 9057 (0.0287%) | 9070 (0.1723%) |
| **5th** | 20695.2 | 20694 (0.0058%) | 20694 (0.0058%) | 20694 (0.0058%) | 20694 (0.0058%) |

**Table 14. DFT Calculation times**

| DFT Execution Time (Complex Phasor) | | | | |
|---|---|---|---|---|
| **Dataset** | **256 Samples** | **128 Samples** | **64 Samples** | **32 Samples** |
| **1st** | 1318 µs | 170 µs | 85 µs | 46 µs |

**Table 14. DFT Calculation times (continued)**

| | | | | |
|---|---|---|---|---|
| 2nd | 1319 µs | 170 µs | 85 µs | 46 µs |
| 3rd | 1312 µs | 170 µs | 85 µs | 46 µs |
| 4th | 1318 µs | 170 µs | 85 µs | 46 µs |
| 5th | 1318 µs | 170 µs | 85 µs | 46 µs |

# 7 Total Harmonic Distortion (THD)

The THD of a signal is shown with the following equation:

THD formula

$$THD \equiv \frac{\sqrt{V_2^2 + V_3^2 + V_4^2 + \ldots + V_n^2}}{V_1} \qquad \textit{Eqn. 17}$$

V1 is the amplitude of the fundamental frequency, and V2, … Vn are the amplitude of the harmonics.

The fundamental RMS voltage was calculated using the following equation:

$$Vrms\_fundamenta = \sqrt{\frac{Vreal^2 + Vimg^2}{2}} \qquad \textit{Eqn. 18}$$

Vreal is the real part of the DFT phasor and Vimg is the imaginary part of the phasor.

## 7.1 Implementation

The input parameters for the THD are two 16-bit unsigned data values. One for the total RMS and another for the fundamental RMS. The output value is a 32-bit unsigned number. The output number is multiplied by 65536 to display the decimal values with a 16-bit resolution.

The THD implemented code is illustrated in Figure 18.

```
//-------------------------------------------
//   THD
//-------------------------------------------
word THD_calc(word Total_RMS, word Fund_RMS){

  if(Total_RMS >= Fund_RMS)
    return( (((dword)(SquareRoot((Total_RMS*Total_RMS) - (Fund_RMS*Fund_RMS))))<<16) / (Fund_RMS));
  else
    return(0);


}
```

**Figure 18. THD implementation**

The input for the fundamental RMS algorithms are the DFT phasors that consist of two 32-bit signed values. Its implementation is shown in Figure 13. The output parameter is a 16-bit unsigned value.

The fundamental RMS implemented code is illustrated in Figure 19.

```
//-------------------------------------------
//   RMS DFT
//-------------------------------------------
word RMS_DFT_calc(Complex A){

  return(SquareRoot((((dword)A.Real*A.Real + (dword)A.Img*A.Img))/2));

}
```

**Figure 19. Fundamental RMS calculations**

## 7.2 Test Result

The THD and fundamental RMS performance were evaluated with the same datasets used for the DFT. Please refer to Section 6.2, “Test Result” for the dataset reference.

The results are presented in Table 16.

To calculate the THD and fundamental RMS the following inputs are used:

- DFT phasor of the fundamental frequency
- RMS value calculated by equation 3

The THD and fundamental RMS performance are not affected directly by the number of samples used in the input buffer but are affected by the precision of the input parameters.

The precision of these input parameters affect the outputs precision. (THD and fundamental RMS)

The tests were performed using the DFT phasor and RMS value, both were calculated with 256 samples for the maximum precision of the input values. For information regarding the DFT and RMS algorithms precision, please refer to Section 6, “ Discrete Fourier Transform (DFT)” and Section 4, “RMS Calculation .”

**Table 15. THD and fundamental RMS test results**

| THD and DFT RMS (DFT with 256 points) | | | | | |
|---|---|---|---|---|---|
| | **1st dataset** | **2nd dataset** | **3rd dataset** | **4th dataset** | **5th dataset** |
| **Vrms fund expected** | 23170 | 23170 | 10263 | 10214 | 14748 |
| **Vrms fund obtained** | 23168 | 23169 | 10262 | 10213 | 14746 |
| **Vrms fund error** | 0.00763% | 0.00331% | 0.00789% | 0.01042% | 0.01052% |
| **THD expected** | 0 | 0 | 30071 | 30887 | 27577 |
| **THD obtained** | 608 | 0 | 30072 | 30891 | 27581 |
| **THD error** | — | — | –0.00447% | –0.01393% | –0.01489% |
| **execution time [uS]** | 65 | 44 | 72 | 80 | 72 |

# 8 Conclusions

The goal of this application note is to supply information regarding the MCF51EM256 capability for processing algorithms commonly used in metering with a quantitative approach. Several algorithms were implemented and its execution time and precision measured.

As per the double buffer approached for storing the input sampled data, the time of filling one buffer is the time available for processing the other buffer. In this application note a buffer is considered to be filled within 16.667 ms. Each buffer contains a full period of a 60 Hz sine wave. Therefore, the MCF51EM256 would have less then 16.667 ms to do all the algorithm calculations in a set of 256 samples per phase. Some portion of this time should be left for the other application functionalities, as updating the LCD, managing the user interface, manage eventual communications protocols, perform data normalization, and others.

Table 16 illustrates the processing capability of the MCF51EM256 for implementing a 3-phase metering system.

**Table 16. Summary of MCF51EM256 performance**

| Algorithm | Execution Time [uS] | # per phase | # of phases | Total | % of total available time |
|---|---|---|---|---|---|
| Energy per phase | 610 | 1 | 3 | 1950 | 11.7% |
| Active power per phase | 3 | 1 | 3 | 9 | 0.1% |
| Total RMS per signal with 256 samples | 280 | 0 | 3 | 0 | 0.0% |
| Total RMS per signal with 128 samples | 170 | 0 | 3 | 0 | 0.0% |
| Total RMS per signal with 64 samples | 110 | 2 | 3 | 780 | 4.7% |
| Total RMS per signal with 32 samples | 80 | 0 | 3 | 0 | 0.0% |
| Apparent Power | 1 | 1 | 3 | 3 | 0.0% |
| Reactive Power | 40 | 1 | 3 | 120 | 0.7% |
| Power Factor | 90 | 1 | 3 | 270 | 1.6% |
| DFT per signal (256 samples) per signal | 1320 | 0 | 3 | 0 | 0.0% |
| DFT per signal (128 samples) per signal | 170 | 0 | 3 | 0 | 0.0% |
| DFT per signal (64 samples) per signal | 85 | 6 | 3 | 1800 | 10.8% |
| DFT per signal (32 samples) per signal | 46 | 0 | 3 | 0 | 0.0% |
| THD (includes fundamental RMS) per signal | 80 | 1 | 3 | 240 | 1.4% |
| Application free time | | | | 11495 | 69.0% |
| | | Total metering algorithms time | | 5172 | 31.0% |
| | | Total time | | 16667 | 100.0% |

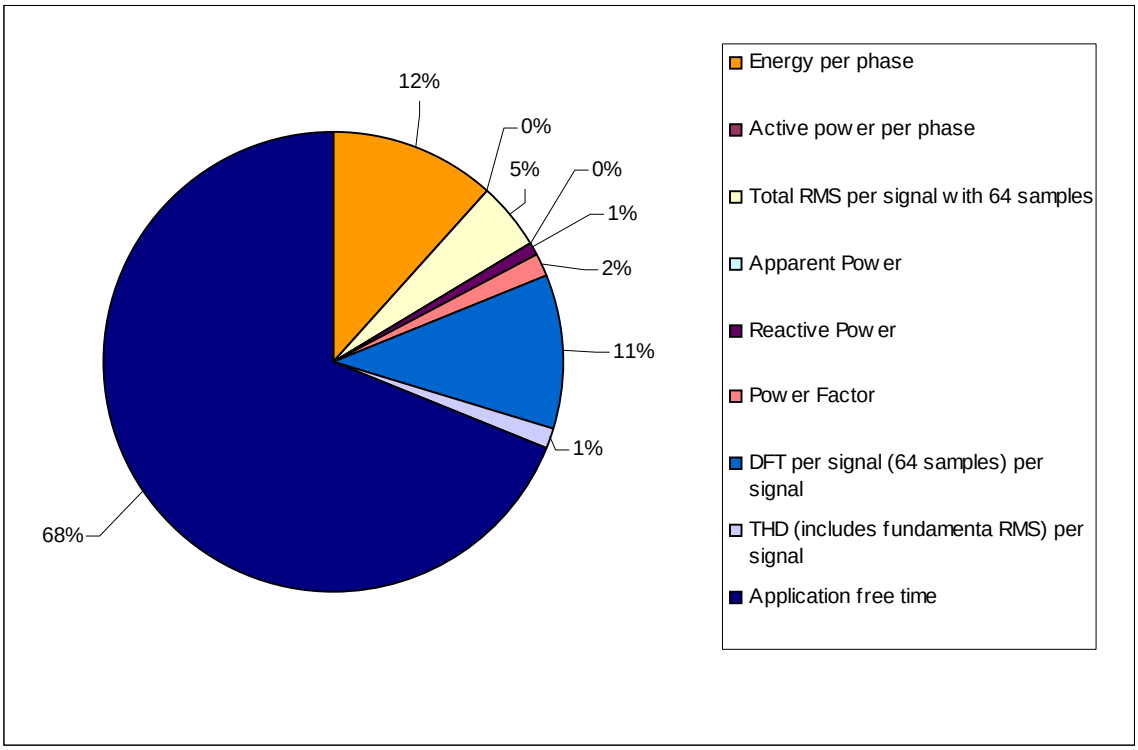


**Figure 20. Percentage of CPU performance used for implemented algorithms**

THIS PAGE IS INTENTIONALLY BLANK

## *How to Reach Us:*


**Home Page:**
www.freescale.com

**Web Support:**
http://www.freescale.com/support

**USA/Europe or Locations Not Listed:**
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

**Europe, Middle East, and Africa:**
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

**Japan:**
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

**Asia/Pacific:**
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

***For Literature Requests Only:***
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or +1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com








AN3896
Rev. 0
010/2009